Game Structure Overview

1 GameManager

The **GameManager** is the central component that manages the overall game loop and coordinates various components like the MVC structure, the Facade, and the Observer-Subject pattern.

- Handles the game loop: input processing, updating game state, and rendering.
- Creates and manages game objects like players, enemies, and initializes MVC components (**Model**, **Controller**, **View**).
- Acts as the central hub that coordinates interactions between different components.

```
class GameManager {
    GameManager() {
        inputHandler  = new InputHandler();
        render        = new Renderer();
        gameState     = new GameStateHandler();
    }
    void run() {
        while (running) {
            input = processInput();
            updateGame(input);
            refresh();
        }
    }
private:
    int processInput() {
        return input->getInput();
    }
    void updateGame(input) {
        gameState->update(input);
    }
    void refresh() {
        render->renderScene(currentScene);
    }
    InputHandler*     inputHandler;
    Renderer*         render;
    GameStateHandler* gameState;
    bool              running = true;
};
```

2 Model

The **Model** represents the core data and logic of the game, such as player stats, enemy behaviors, and game rules. It holds the state of the game world and the logic to manipulate it.

- Holds the game's data (e.g., player's position, health).
- Implements game logic like movement, collisions, and AI behavior.
- Notifies the **Observer** (View) when the model changes using the Observer pattern.

```
class Paddle {
public:
    int x, y;
    Player() : x(0), y(0) {}
    void moveLeft() { x--; }
    void moveRight() { x++; }
};
```

3 MVC Structure

The MVC (**Model**, **View**, and **Controller**) is the core of the game architecture. It separates concerns by dividing the application into three distinct parts: the data (Model), the user input (Controller), and the presentation (View).

- **Model** - Holds game data and logic.
- **Controller** - Handles user input and updates the Model.
- **View** - Renders the game world based on the Model.

```
class Controller {
public:
    int getInput() {
        if (keyPressed(LEFT)) return LEFT;
        if (keyPressed(RIGHT)) return RIGHT;
    }
};
```

4 Facade

The **Facade** provides a simplified interface to the underlying complex game logic. It contains higher-level game operations that interact with the Model and the Controller.

- Encapsulates the game logic (e.g., moving the player, updating enemies).
- Makes it easier to manage complex operations by exposing a simplified interface.

```
class GameFacade {
public:
    void updateGameLogic(Paddle& paddle, Ball& ball) {
        paddle.moveLeft();   // Example of movement logic
        ball.update();     // Example of enemy AI logic
    }
};
```

5 Observer-Subject Pattern

The **Observer-Subject** pattern is used to allow the **View** to be notified whenever the **Model** changes (e.g., player's position, health). This ensures that the view is always up-to-date with the game state without the need for constant polling.

- The **Model** (Subject) notifies the **View** (Observer) of state changes.
- The **View** updates the display when the model changes.

```
class Subject {
    public:
        void addObserver(Observer* observer) {
            observers.push_back(observer);
        }

        void notifyObservers() {
            for (auto& observer : observers)
                observer->update();  // Notify view
        }

    private:
        std::vector<Observer*> observers;
};

class Player : public Subject {
    public:
        void moveUp() {
            y--;
            notifyObservers();  // Notify view when player moves
        }
};

class View : public Observer {
    public:
        void update() override {
            renderScene();  // Update view when notified
        }
};
```

6 Inheritance

Inheritance is a core concept in object-oriented programming where a class (child) derives from another class (parent) and inherits its properties and methods. This establishes an "is-a" relationship between the child and parent classes. However, inheritance can lead to tightly coupled code and deep hierarchies, making the system harder to maintain and extend.

```
class Animal {
public:
    virtual void sound() = 0;
};

class Dog : public Animal {
public:
    void sound() override { std::cout << "Bark"; }
};
```

7 Composition

Composition is a design principle where objects are composed of other objects to achieve complex functionality. This follows the "has-a" relationship, where one class contains another. Composition provides more flexibility, as components can be easily swapped or modified at runtime without modifying the entire class structure.

```
class SoundBehavior {
public:
    virtual void sound() = 0;
};

class BarkBehavior : public SoundBehavior {
public:
    void sound() override { std::cout << "Bark"; }
};

class Dog {
private:
    SoundBehavior* soundBehavior;
public:
    Dog(SoundBehavior* sb) : soundBehavior(sb) {}
    void makeSound() { soundBehavior->sound(); }
};
```